

# **L10-11 Recap: fundamental lesson(s)**

- **We discussed what hardware is required to execute an instruction, as well as how to best “organize” it.**

# L10-11 Recap: why it's important...

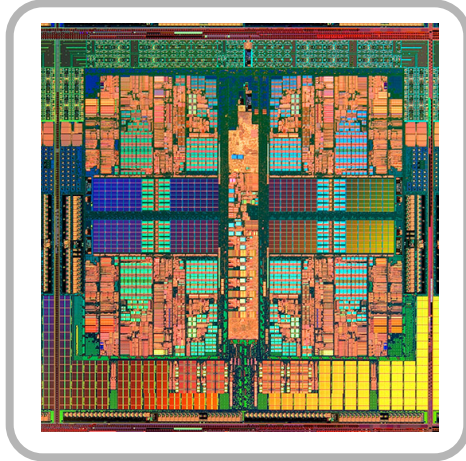
- If you ever design the HW for a microprocessor, etc. you'll need to be aware of these types of issues
  - Much more detail in **Computer Architecture II**
- Understanding organization – and how it impacts the delay of something like a memory reference – will make you a better programmer
  - This will become very clear after the midterm
- It is now more and more important to design HW/SW simultaneously

# **Lecture 12-13: Pipelines**

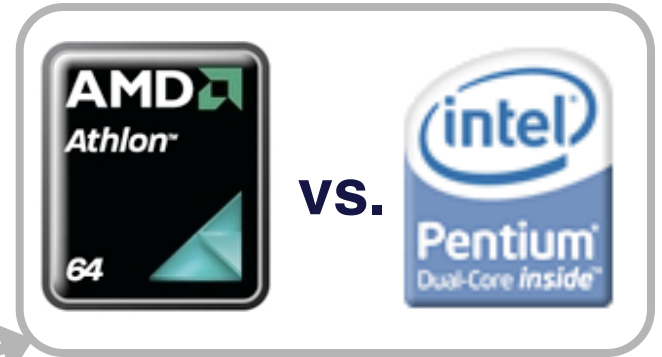
**Suggested reading:  
(HP Chapter 4.5—4.7)**

# Processor components

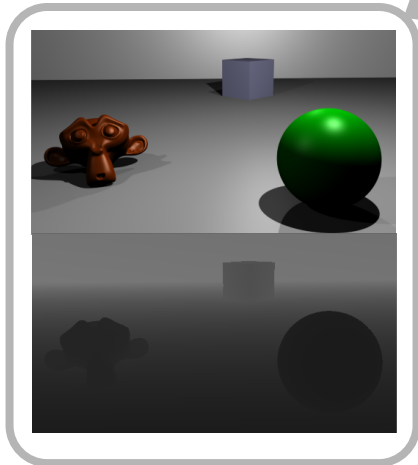
## Multicore processors and programming



## Processor comparison



**Goal:** describe the fundamental components required in a single core of a modern microprocessor as well as how they interact with each other, with main memory, and with external storage media.



Writing more efficient code



The right HW for the right application

```
for i=0; i<5; i++ {  
    a = (a*b) + c;  
}
```

↓

```
MULT r1,r2,r3 # r1 ← r2*r3  
ADD r2,r1,r4  ↓ # r2 ← r1+r4
```

110011	000001	000010	000011
001110	000010	000001	000100

HLL code translation

# Fundamental lesson(s)

- In this lecture, we consider how we can improve performance via higher throughput
  - A classic example is the assembly line
  - If a task as a whole is slow, if it can be broken up into smaller parts – each taking  $\tau$  seconds
  - The time *per task* approaches  $\tau$  – after the pipeline is filled

# Why it's important...

- **Without pipelining, the effective time to do nearly any information processing task would be 5-20 times longer...**

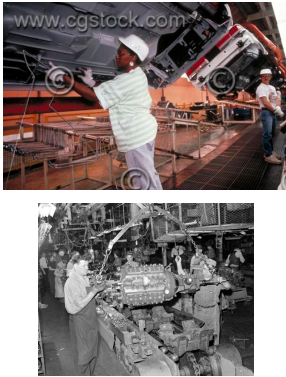
# Example: We want to build N cars...

...Each car takes 6 steps to build...

Build the frame  
(~1 hour)



Build the body  
(~1.25 hours)



Install interior  
(~1.25 hours)



Put on axles, wheels  
(~1 hour)



Paint  
(~1.5 hours)



Roll out  
(~1 hours)





# Sequential Car Building... (a lot like multi-cycle)

Build the frame  
(~ 1 hour)



Build the body  
(~1.25 hours)



Install interior  
(~1.25 hours)



Put on axles, wheels  
(~1 hour)



Paint  
(~1.5 hours)



Roll out (~1 hours)



**Total time: 7 Hours.**  
(~1 hour/stage)

# Pipelined Car Building...

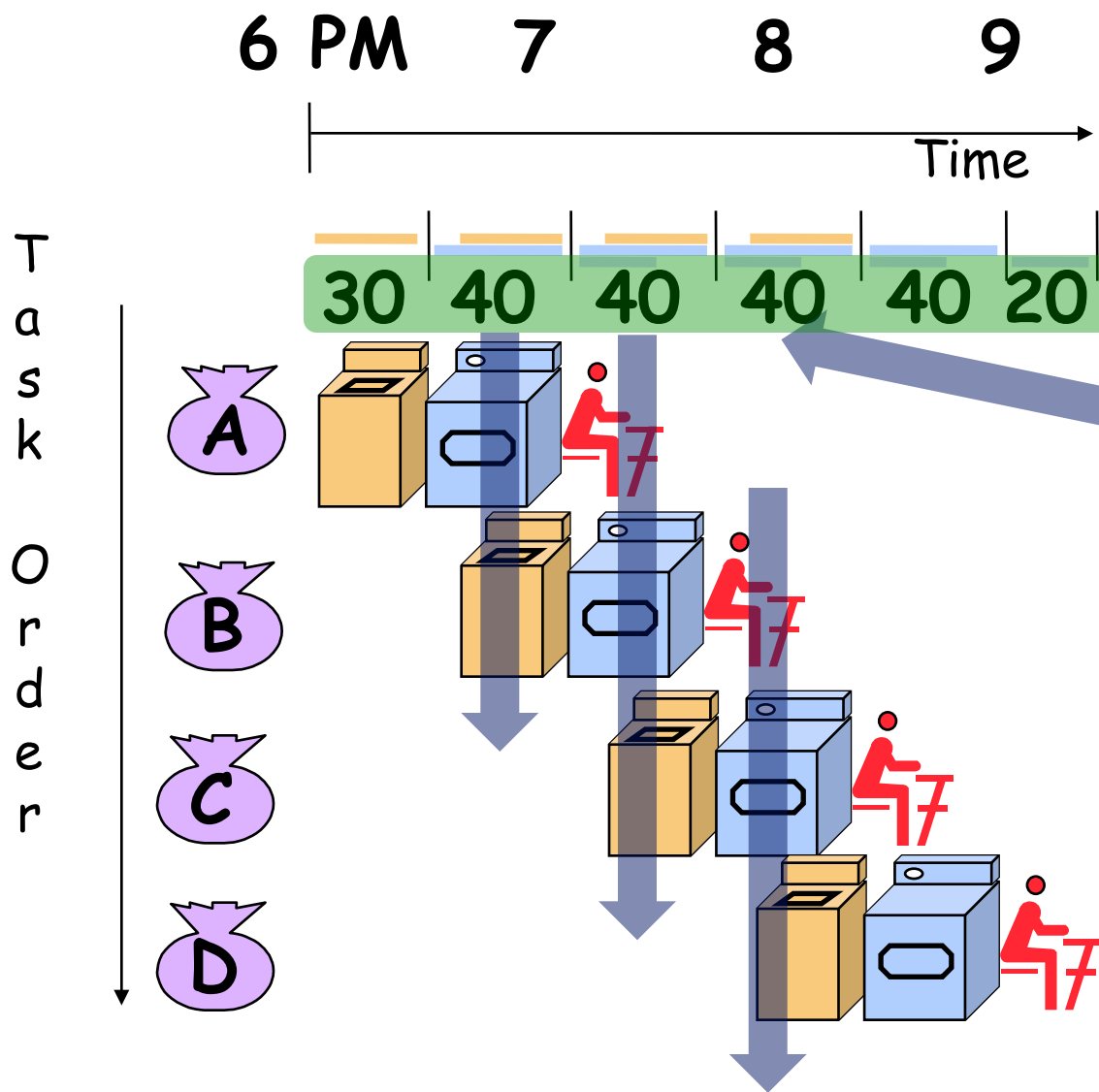


**1 car done ~ every 1.5 hours**

(like multi-cycle, limited by time of the longest stage)



# Pipelining Lessons (laundry example)



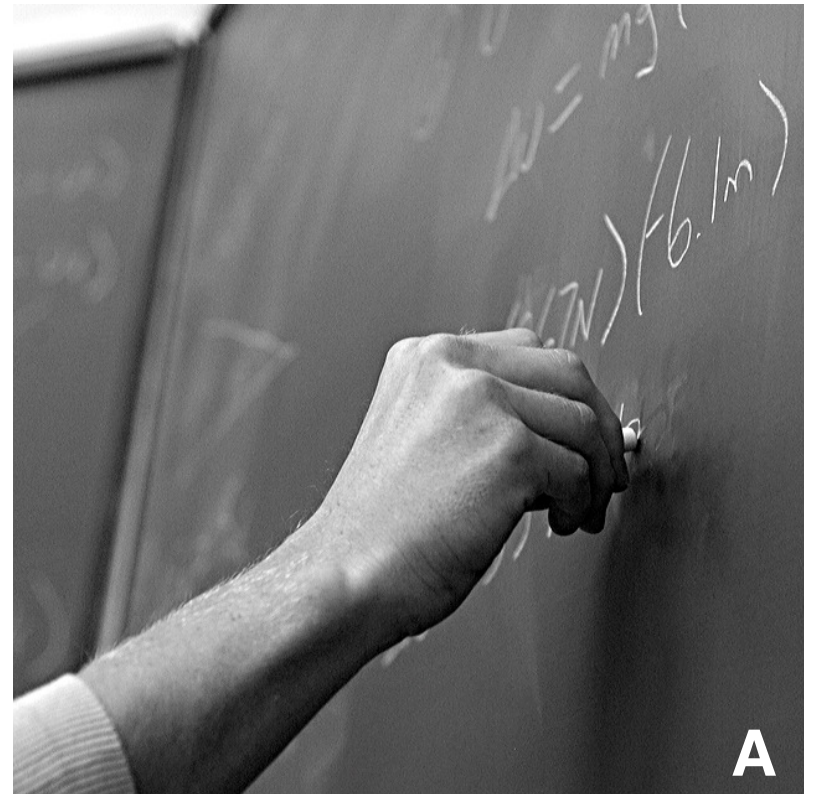
- **Multiple** tasks operating simultaneously
- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Also, need time to “**fill**” and “**drain**” the pipeline.

# Pipelining: Some terms

- If you're doing laundry or implementing a  $\mu\text{P}$ , each stage where something is done called a **pipe stage**
  - In laundry example, washer, dryer, and folding table are pipe stages; clothes enter at one end, exit other
  - In a  $\mu\text{P}$ , instructions enter at one end and have been executed when they leave
- Throughput is how often stuff comes out of a pipeline

# On the board...

- The “math” behind pipelining...



# More detail...

- **Book's approach to draw pipeline timing diagrams...**
  - Time runs left-to-right, in units of stage time
  - Each "row" below corresponds to distinct initiation
  - Boundary between 2 column entries: pipeline register
    - (i.e. laundry basket)
  - Look at columns to see what stage is doing what

0	1	2	3	4	5	6
Wash 1	Dry 1	Fold 1	Pack 1			
	Wash 2	Dry 2	Fold 2	Pack 2		
		Wash 3	Dry 3	Fold 3	Pack 3	
			Wash 4	Dry 4	Fold 4	Pack 4
				Wash 5	Dry 5	Fold 5
					Wash 6	Dry 6

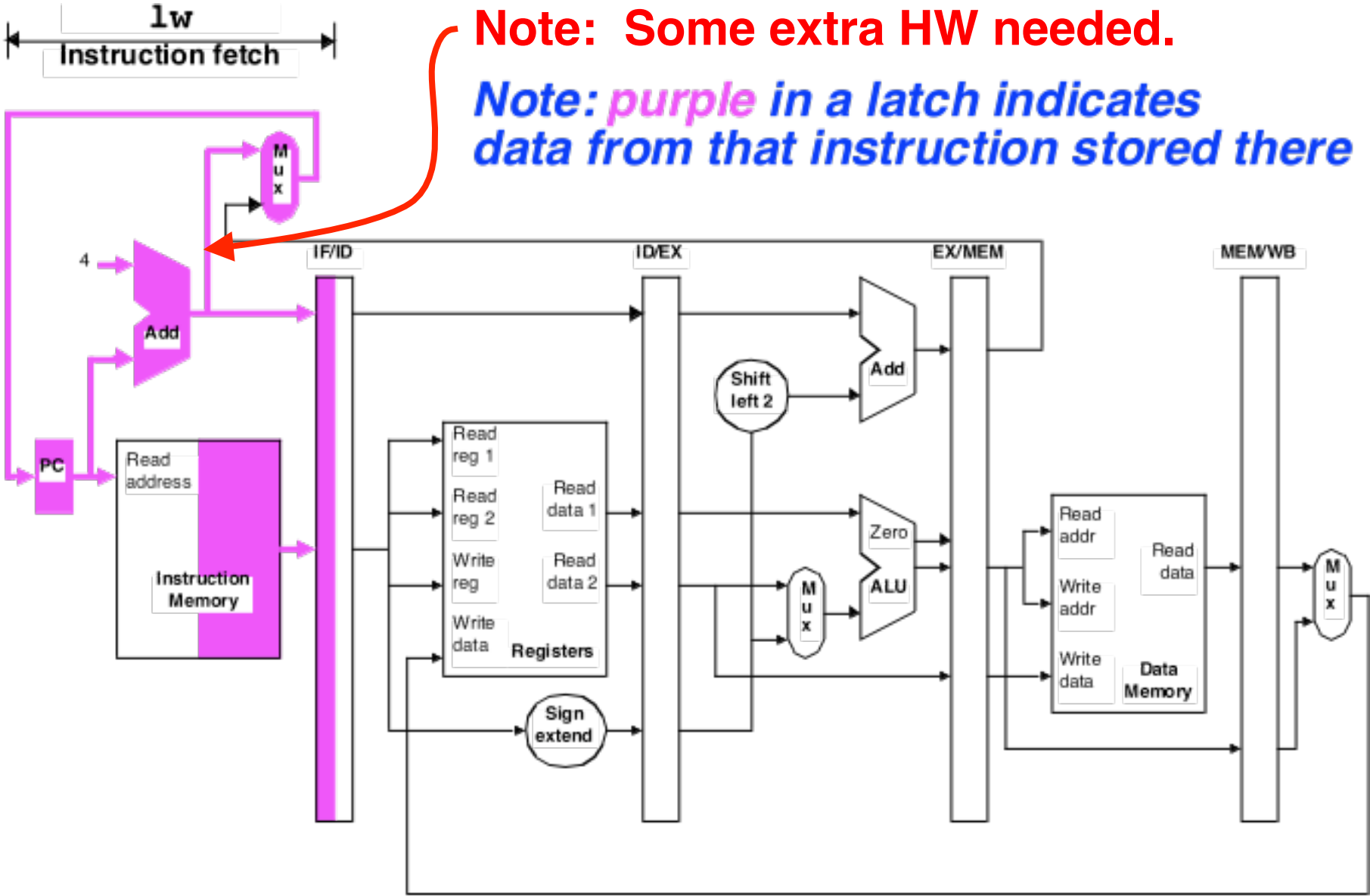
Time for N initiations to complete:

$$NT + (S-1)T$$

Throughput:

$$\text{Time per initiation} = T + (S-1)T/N \rightarrow T!$$

# The “new look” dataflow

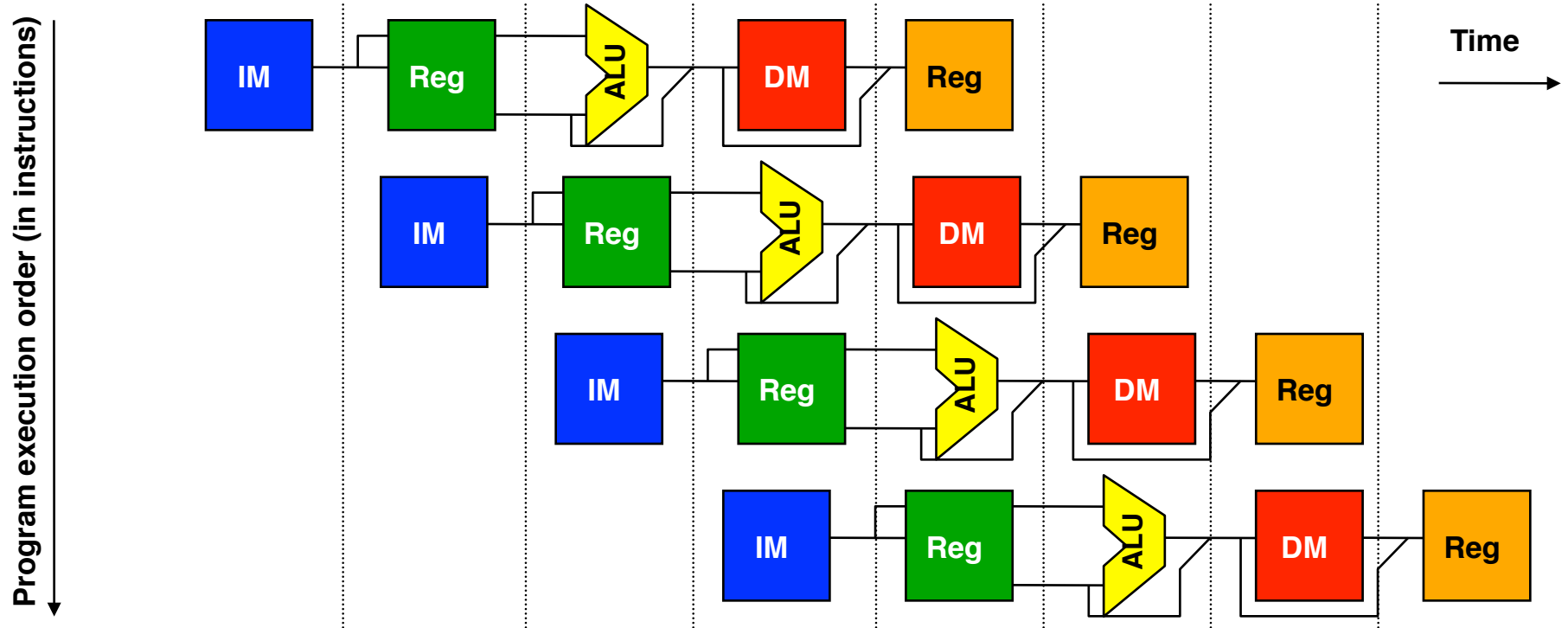


Data must be stored from one stage to the next in pipeline registers/latches.  
Pipeline latch – to – pipeline latch time is one clock cycle

# Another way to look at it...

Clock Number

Inst. #	1	2	3	4	5	6	7	8
Inst. i	IF	ID	EX	MEM	WB			
Inst. i+1		IF	ID	EX	MEM	WB		
Inst. i+2			IF	ID	EX	MEM	WB	
Inst. i+3				IF	ID	EX	MEM	WB



# Impact on instruction execution

- In each cycle, new instruction fetched and begins 5 cycle execution
- In perfect world (pipeline) performance improved 5 times over!
- Now, let's talk about overhead...
  - Must know what's going on in every cycle of machine
  - What if 2 instructions need same resource at same time?
    - (LOTS more on this later)
    - Separate instruction/data memories, multiple register ports, etc. help avoid this



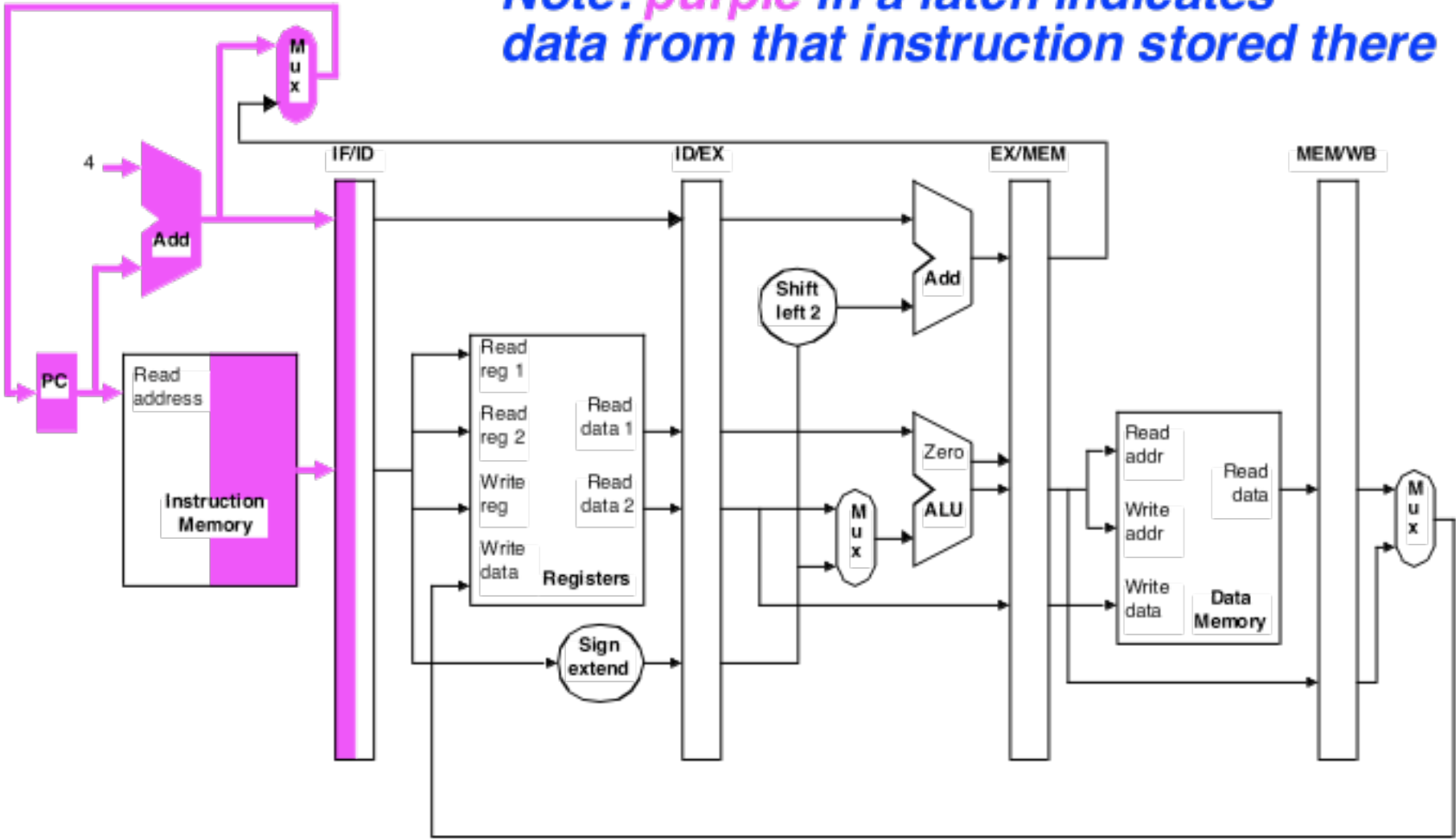
# A preliminary example:

- **Following charts describe 3 scenarios:**
  - **Processing of load word (lw) instruction**
    - **Bug included in design (make SURE you understand the bug)**
  - **Processing of lw**
    - **Bug corrected (make SURE you understand the fix)**
  - **Processing of lw followed in pipeline by sub**
    - **(Sets the stage for discussion of HAZARDS and inter-instruction dependencies)**

# Load word: Cycle 1

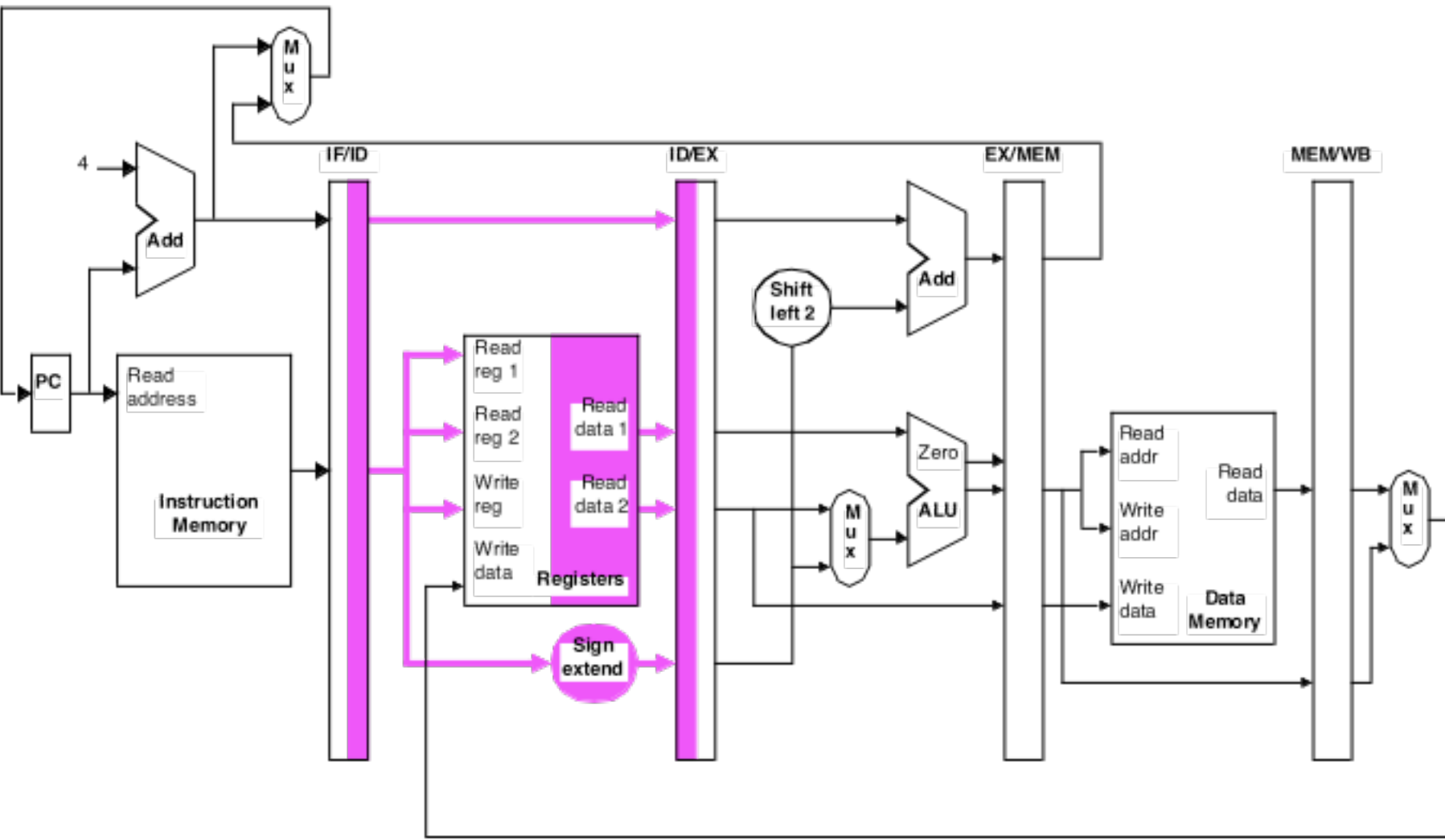
lw  
Instruction fetch

*Note: purple in a latch indicates data from that instruction stored there*



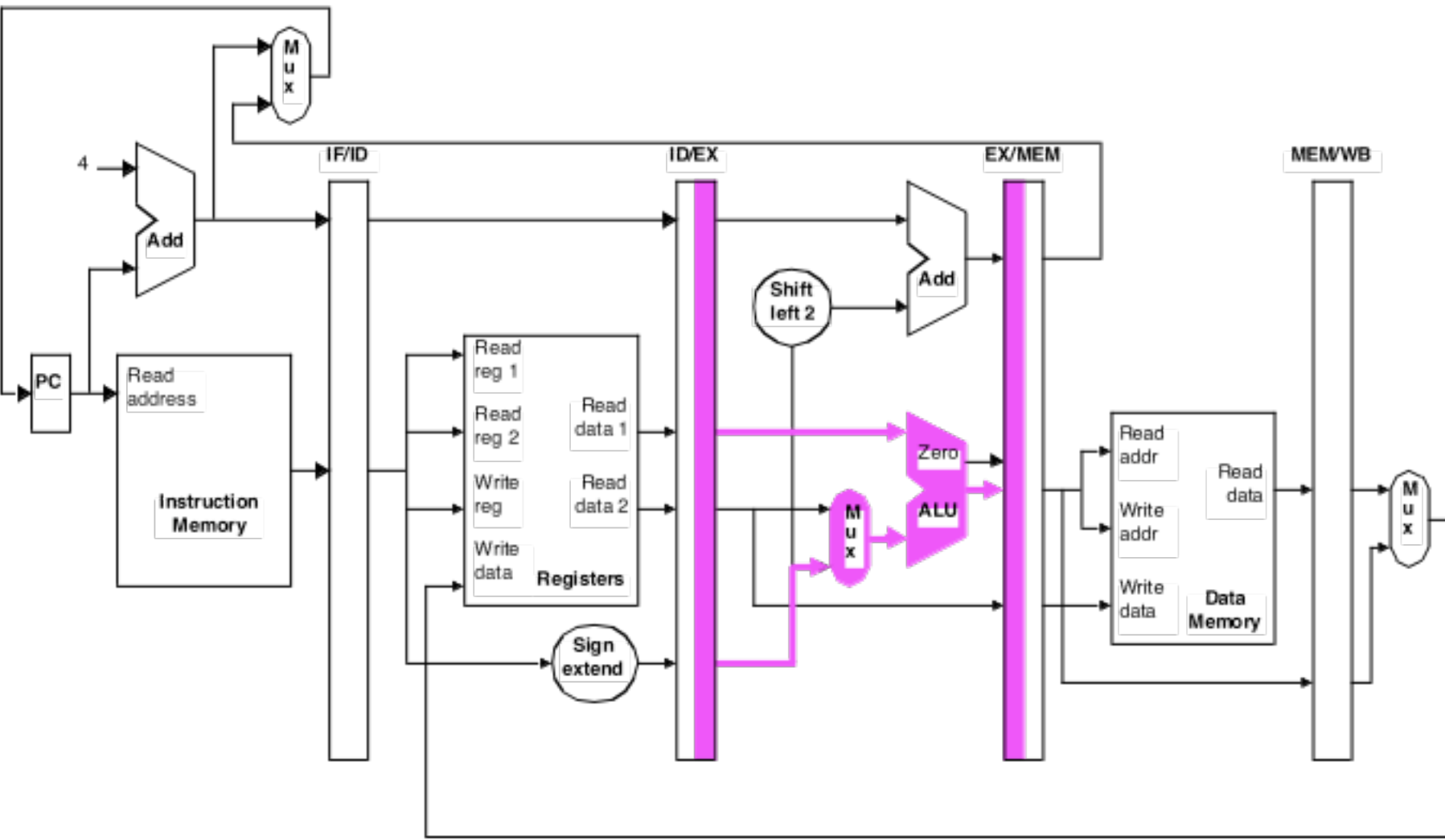
# Load Word: Cycle 2

1w  
Instruction decode



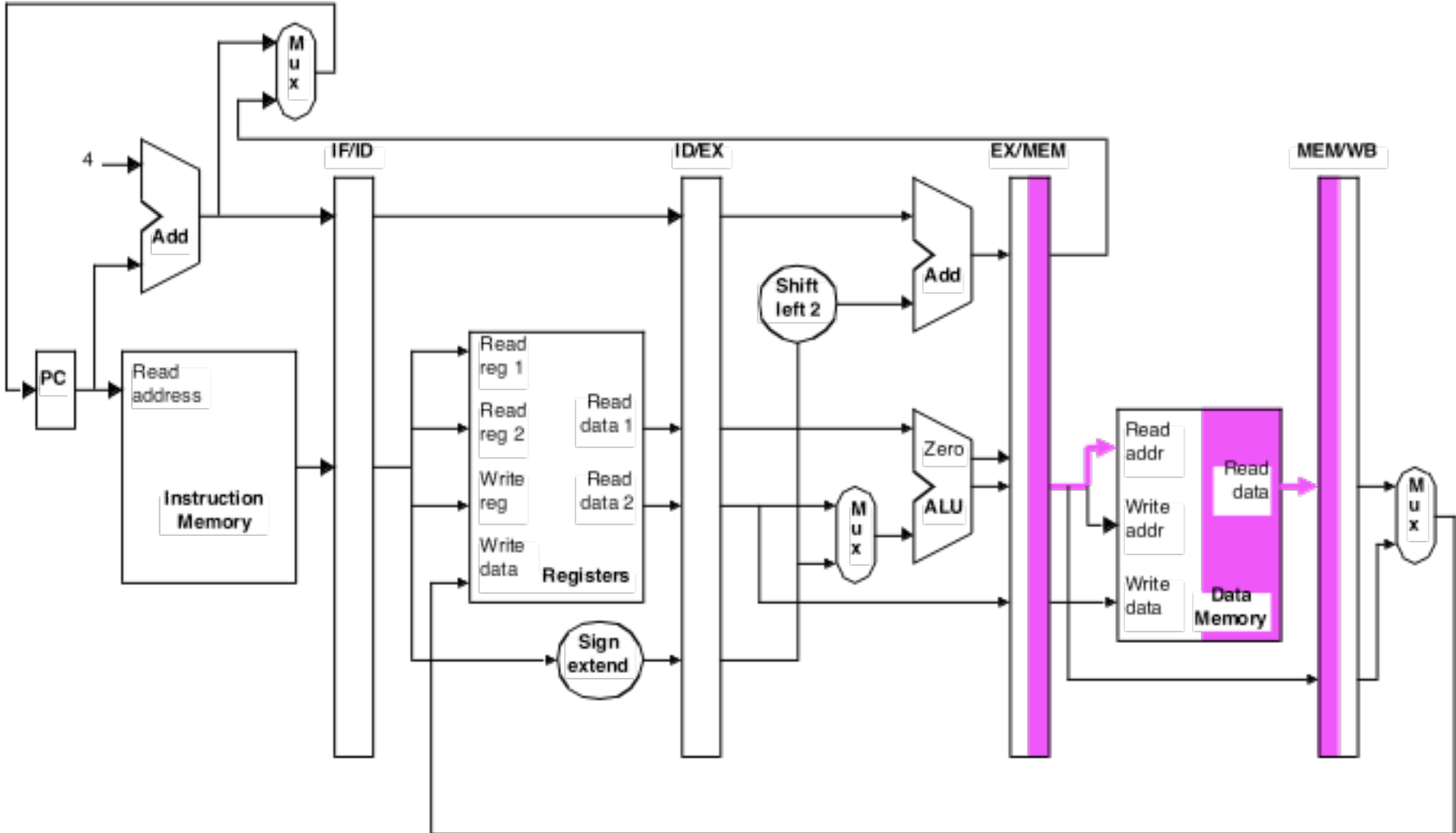
# Load Word: Cycle 3

lw Execution



# Load Word: Cycle 4

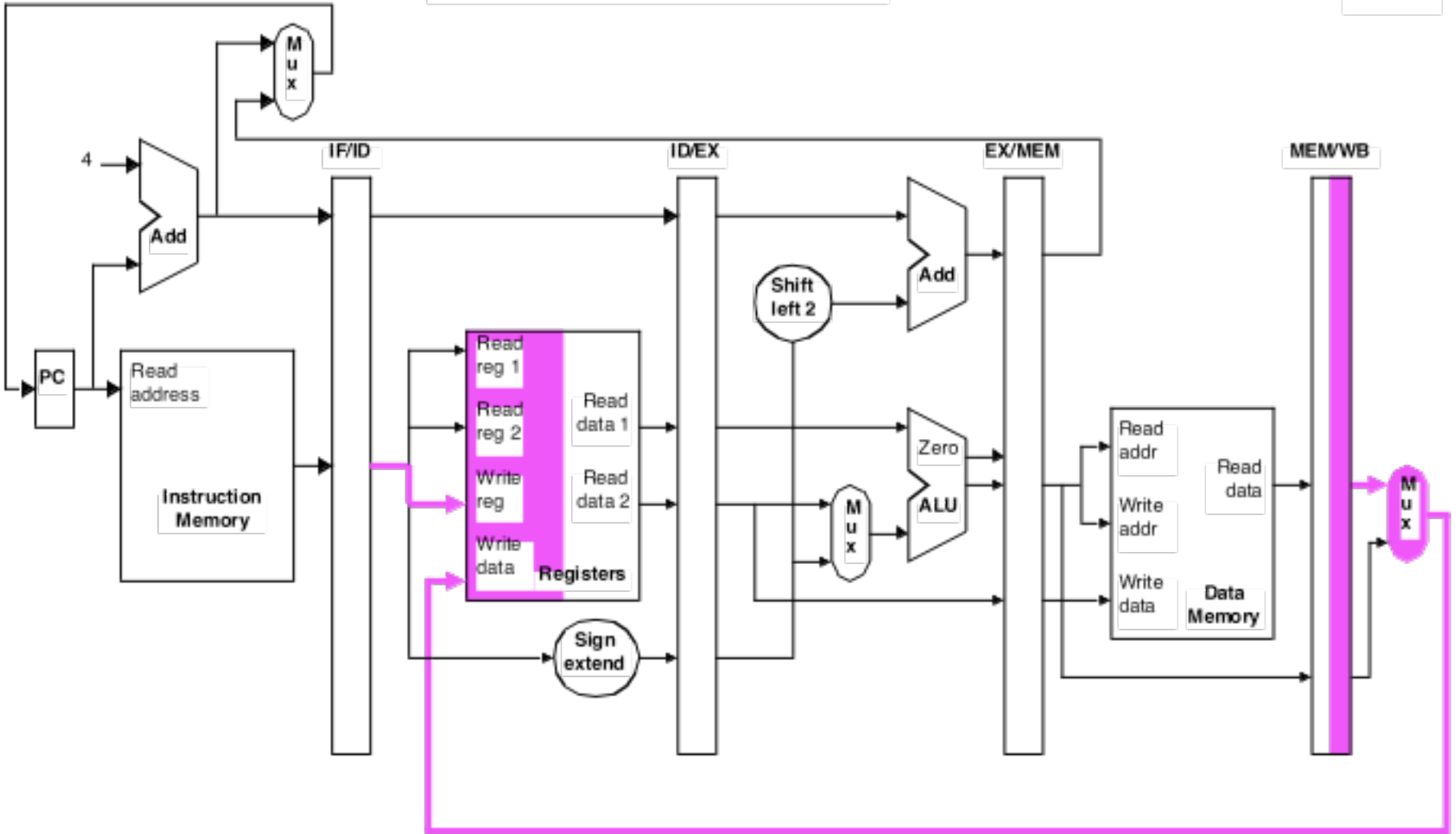
1w  
Memory



# Load Word: Cycle 5

Where's the bug?

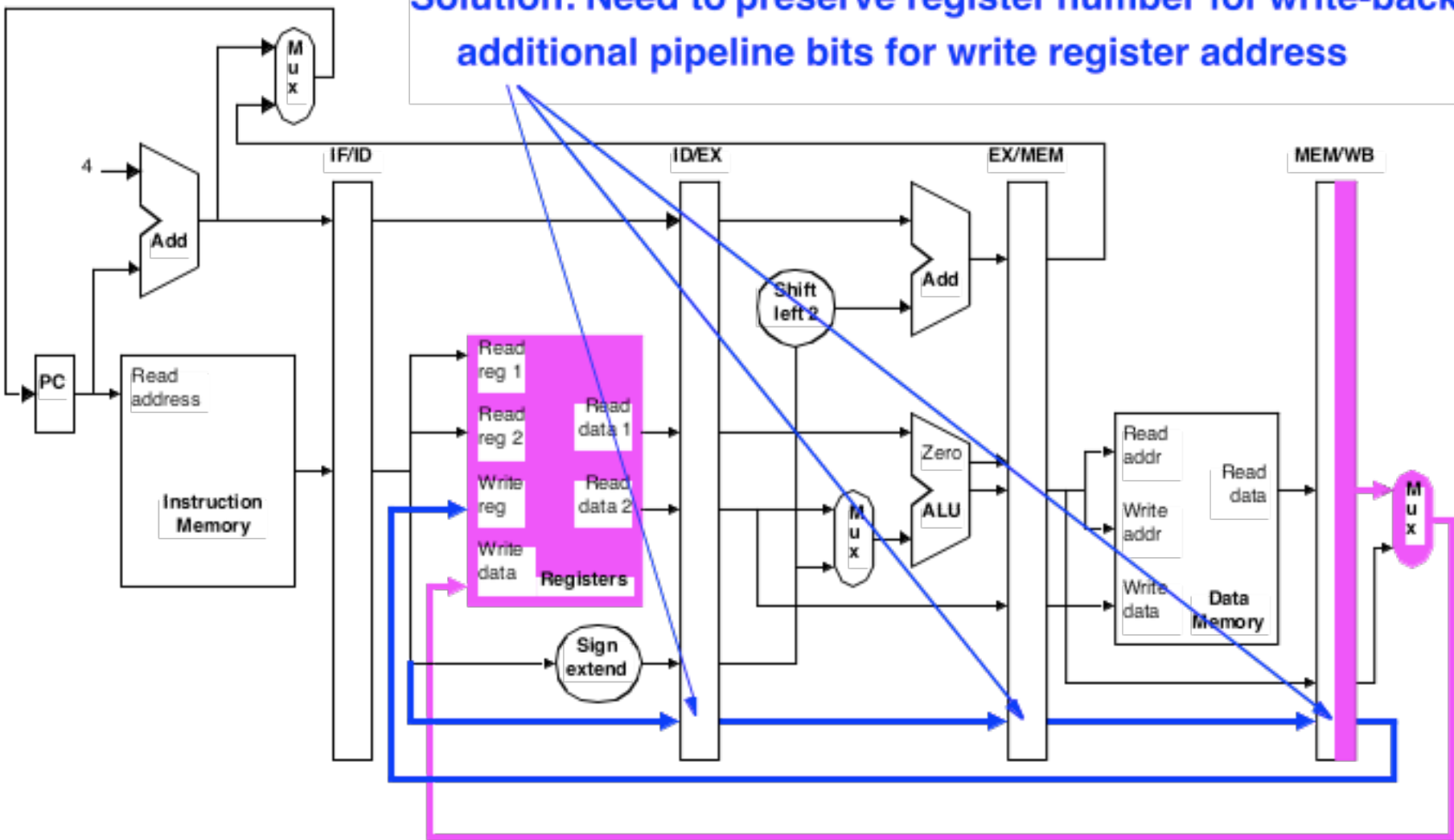
lw  
Write back



# Load Word: Fixed Bug

**Bug:** source for Write Reg is invalid

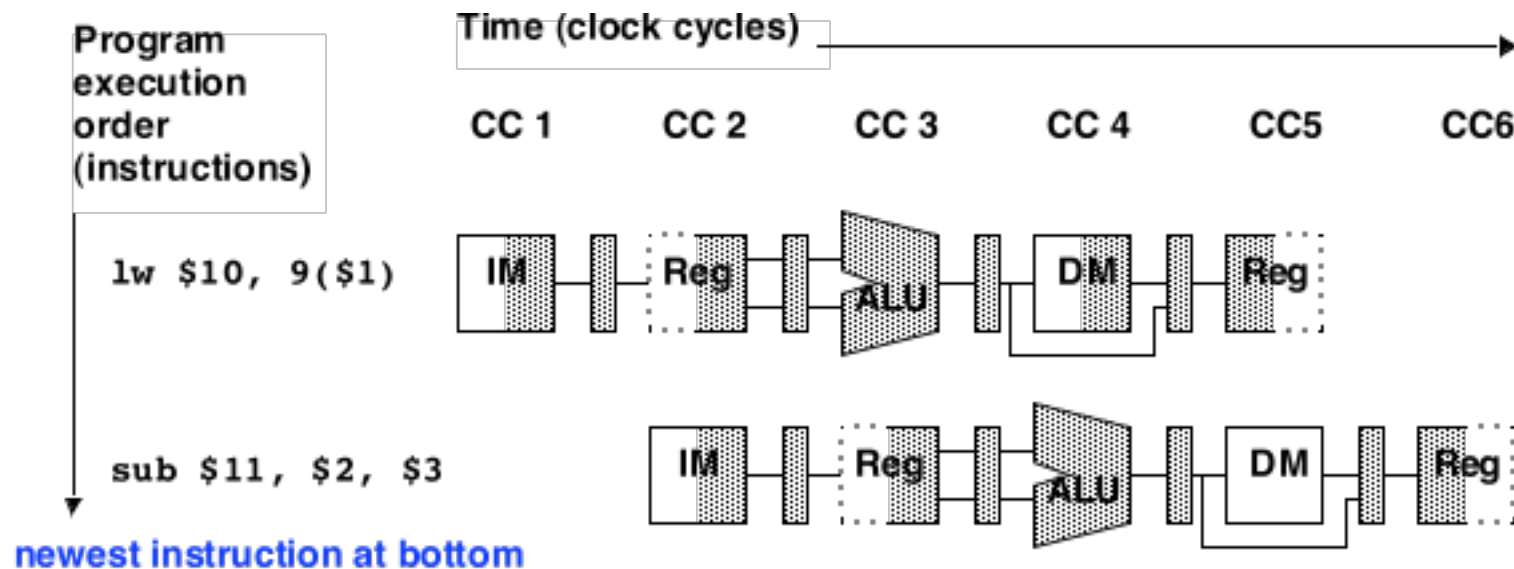
**Solution:** Need to preserve register number for write-back  
additional pipeline bits for write register address



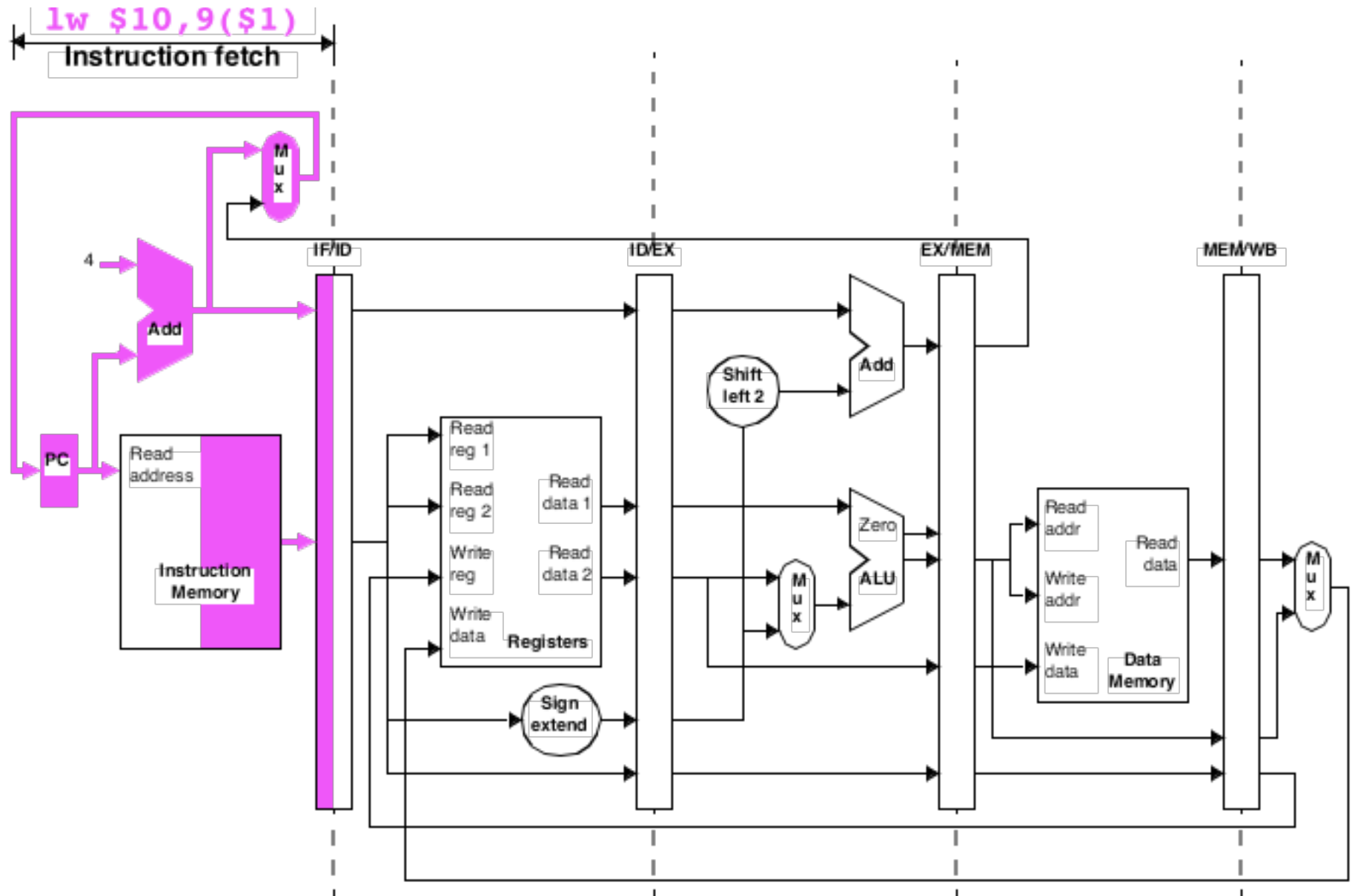


# A 2 instruction sequence

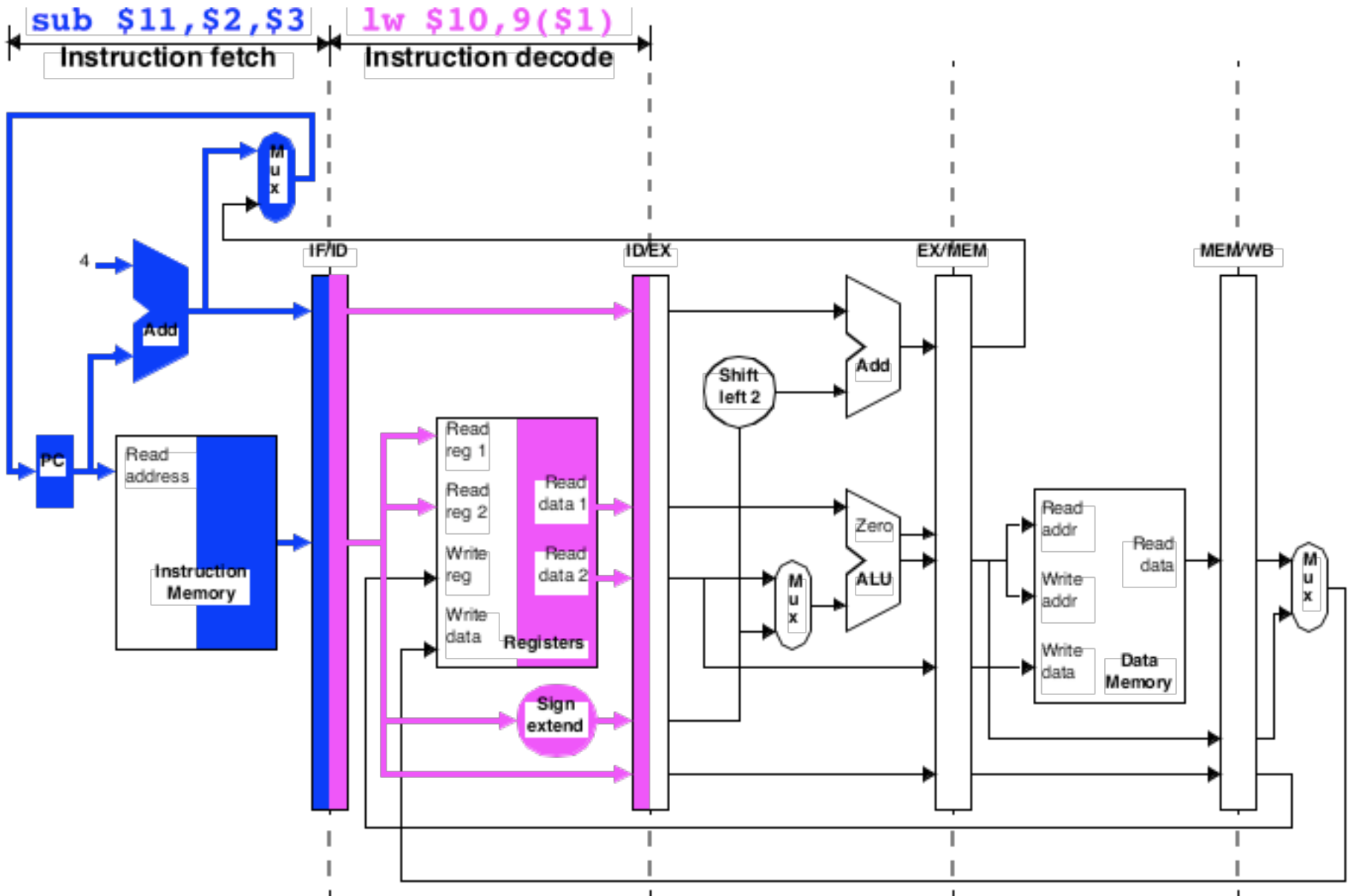
- Examine multiple-cycle & single-cycle diagrams for a sequence of 2 independent instructions
  - (i.e. no common registers b/t them)
    - lw \$10, 9(\$1)
    - sub \$11, \$2, \$3



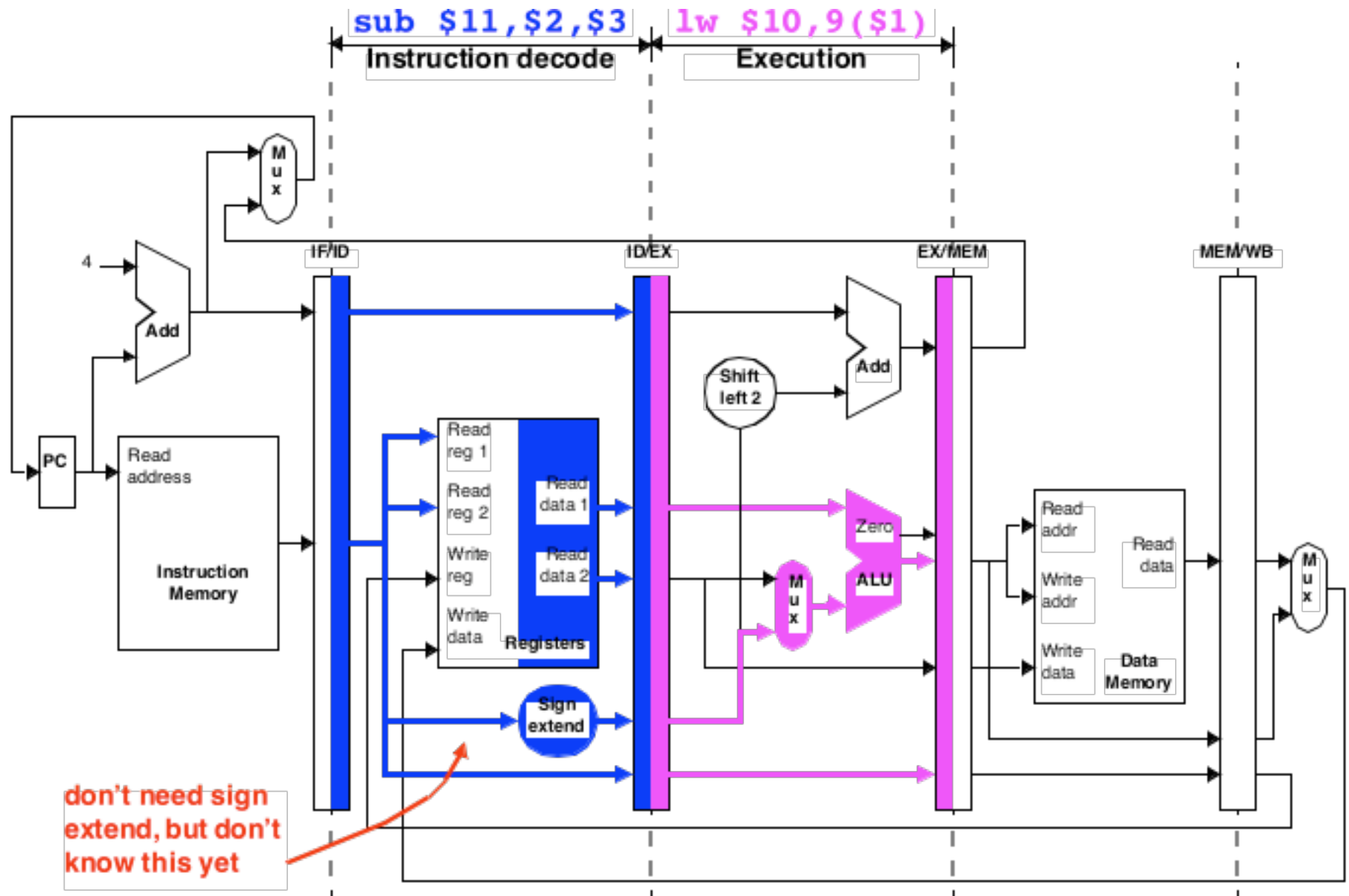
# Single-cycle diagrams: cycle 1



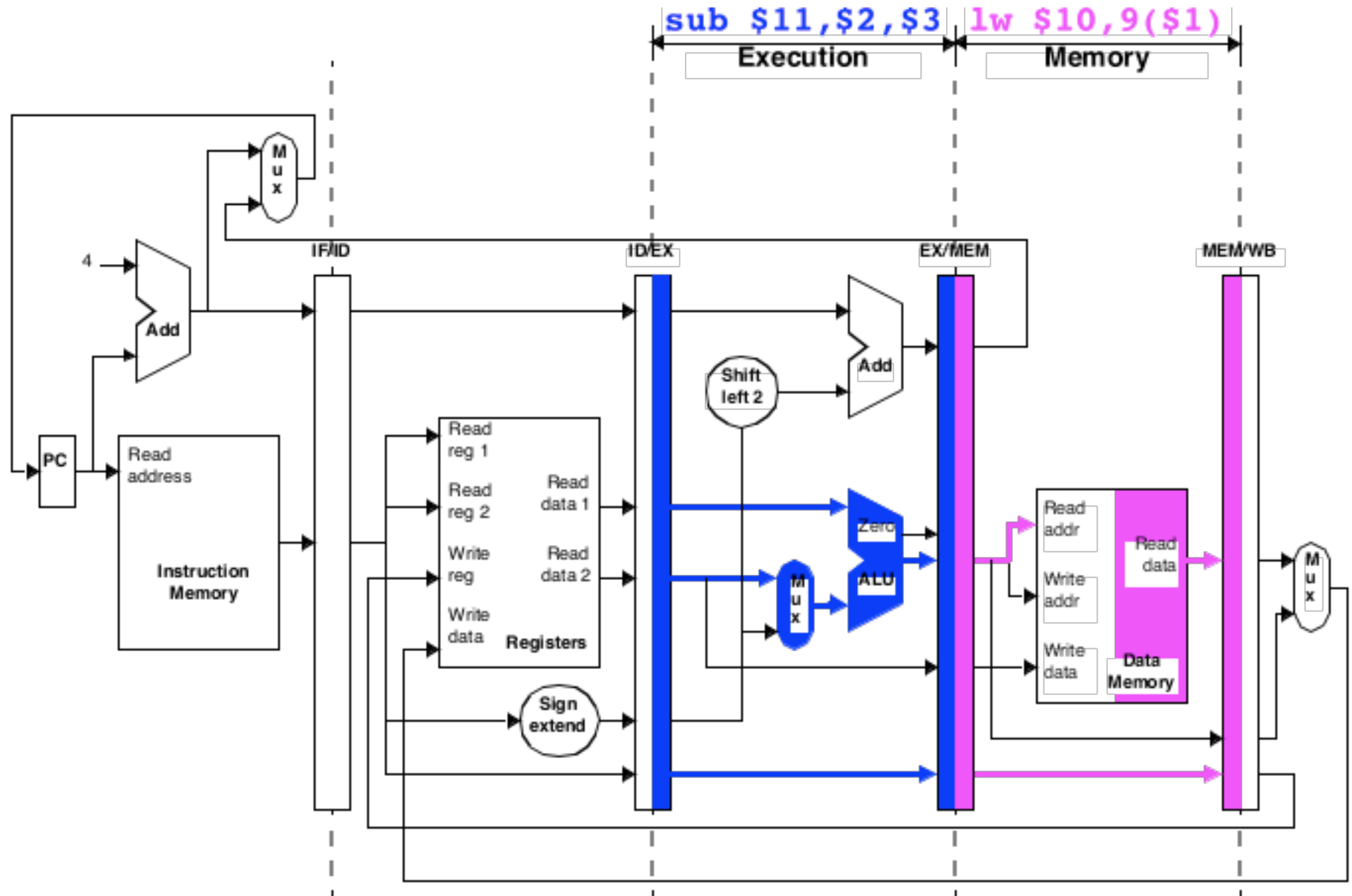
# Single-cycle diagrams: cycle 2



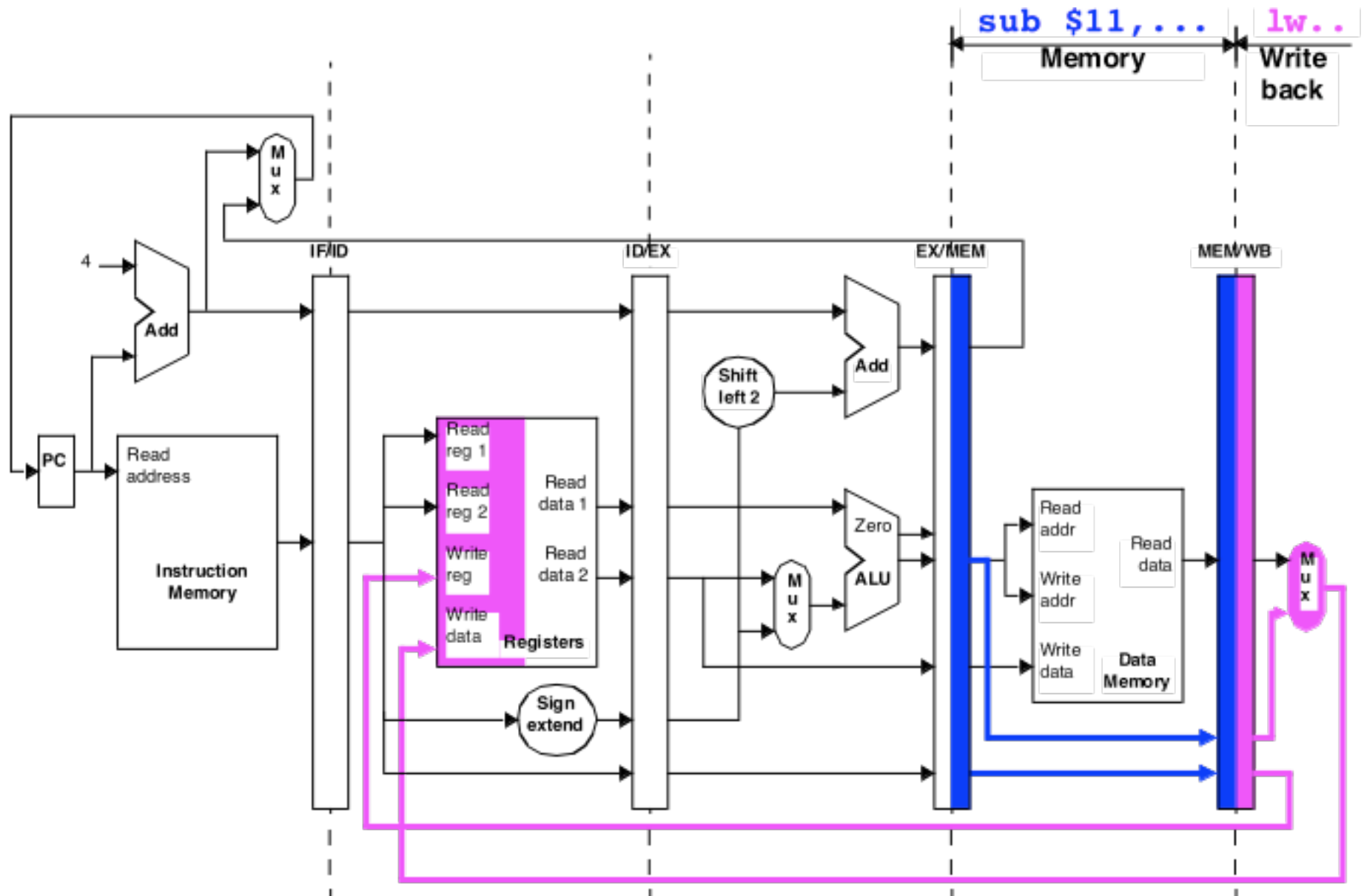
# Single-cycle diagrams: cycle 3



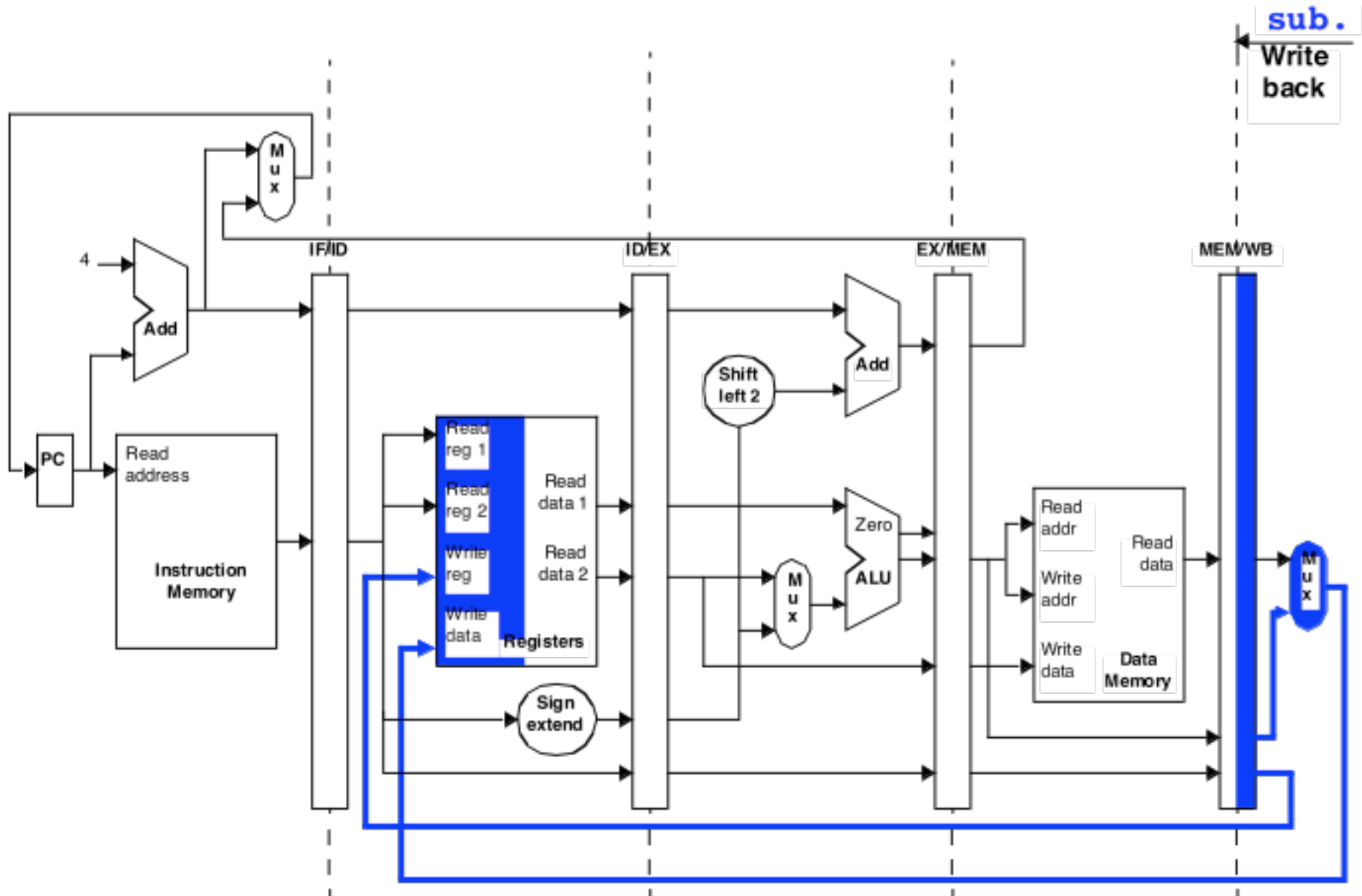
# Single-cycle diagrams: cycle 4



# Single-cycle diagrams: cycle 5



# Single-cycle diagrams: cycle 6





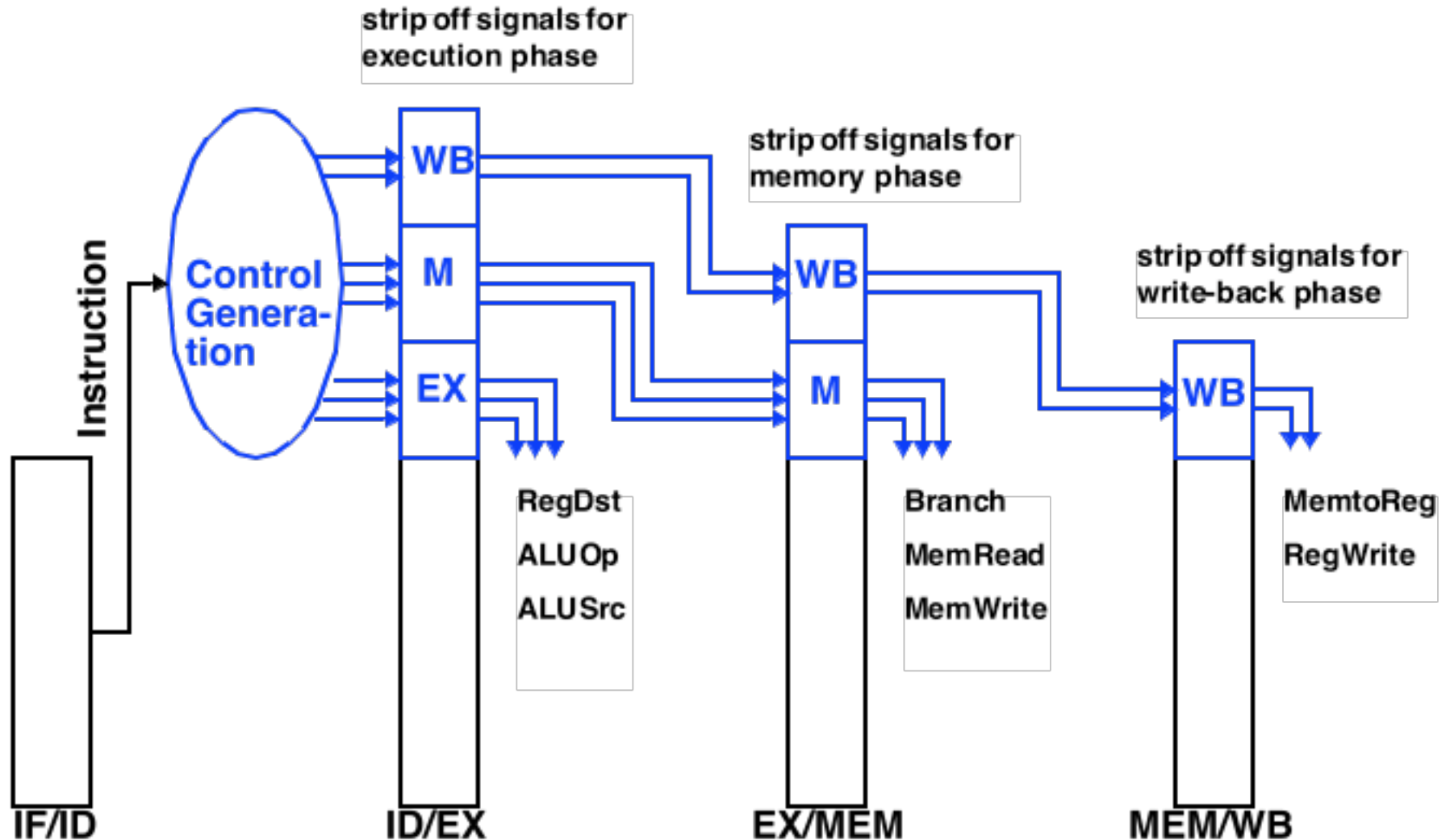
**WHAT ABOUT CONTROL  
SIGNALS?**

# Questions about control signals

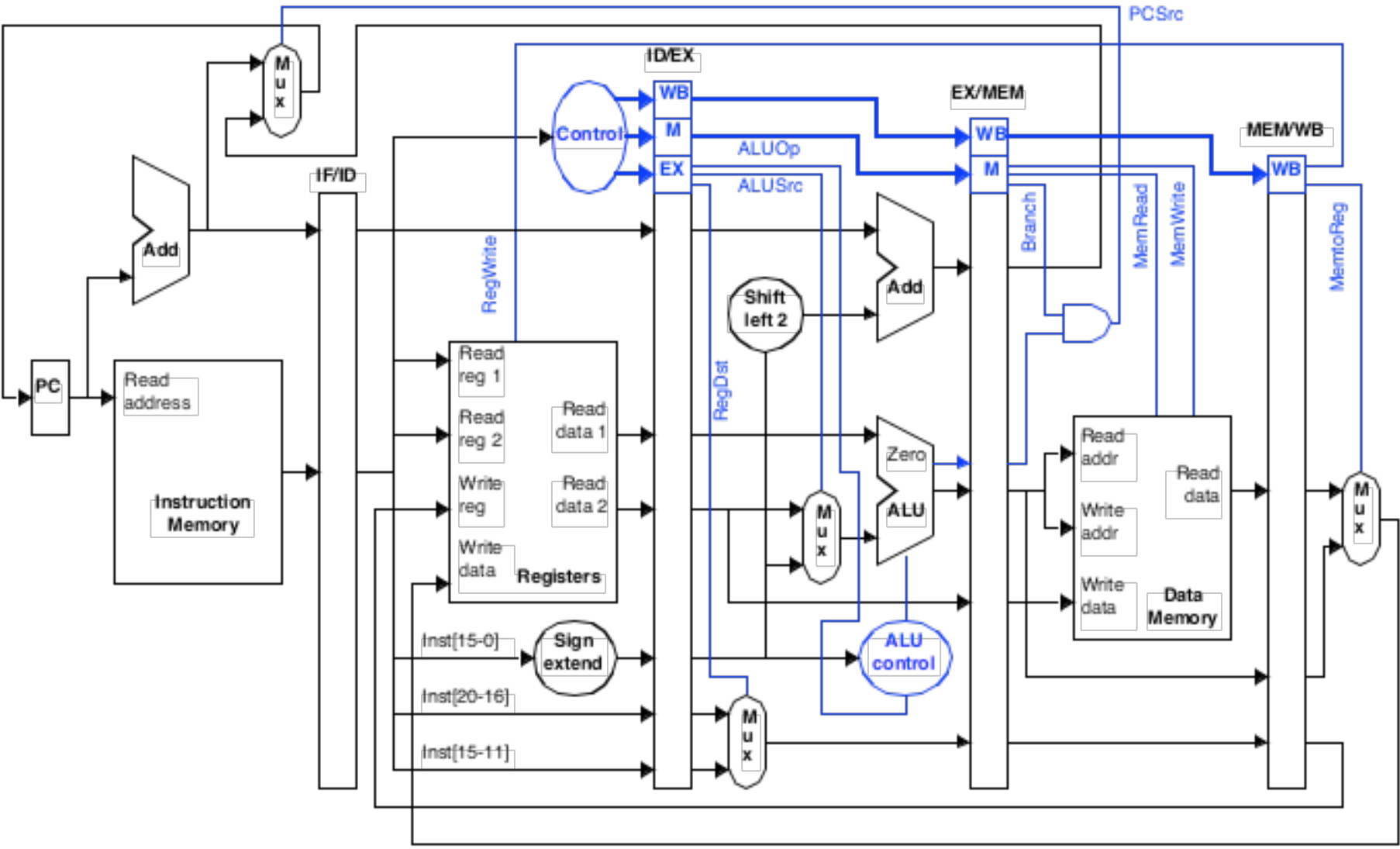
- **Following discussion relevant to a single instruction**
- **Q: Are all control signals active at the same time?**
- **A: ?**
  
- **Q: Can we generate all these signals at the same time?**
- **A: ?**

# Passing control w/pipe registers

- Analogy: send instruction with car on assembly line
  - “Install Corinthian leather interior on car 6 @ stage 3”



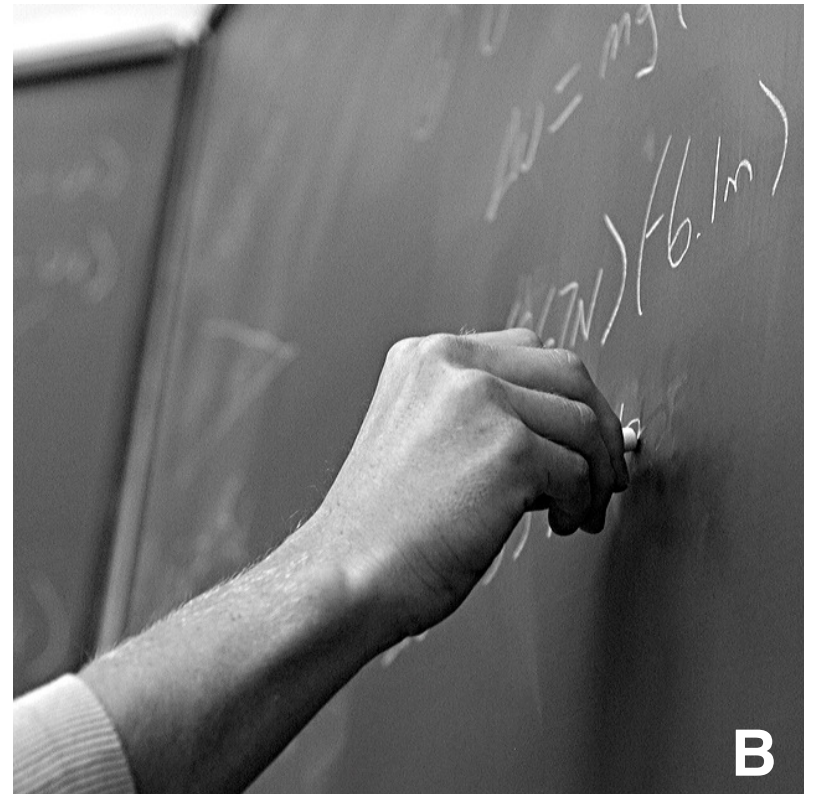
# Pipelined datapath w/control signals



# HAZARDS

# On the board...

- **Let's look at hazards...**
  - **...and how they (generally) impact performance.**



# Pipelining hazards

- **Pipeline hazards prevent next instruction from executing during designated clock cycle**
- **There are 3 classes of hazards:**
  - **Structural Hazards:**
    - **Arise from resource conflicts**
    - **HW cannot support all possible combinations of instructions**
  - **Data Hazards:**
    - **Occur when given instruction depends on data from an instruction ahead of it in pipeline**
  - **Control Hazards:**
    - **Result from branch, other instructions that change flow of program (i.e. change PC)**

# How do we deal with hazards?

- Often, pipeline must be stalled
- Stalling pipeline usually lets some instruction(s) in pipeline proceed, another/others wait for data, resource, etc.
- A note on terminology:
  - If we say an instruction was “**issued later than instruction x**”, we mean that it was issued **after instruction x** and is not as far along in the pipeline
  - If we say an instruction was “**issued earlier than instruction x**”, we mean that it was issued **before instruction x** and is further along in the pipeline



# Stalls and performance

- **Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle**
- **Pipelining can be viewed to:**
  - **Decrease CPI or clock cycle time for instruction**
  - **Let's see what affect stalls have on CPI...**
- **CPI pipelined =**
  - **Ideal CPI + Pipeline stall cycles per instruction**
  - **1 + Pipeline stall cycles per instruction**

# Stalls and performance

- Ignoring overhead and assuming stages are balanced:

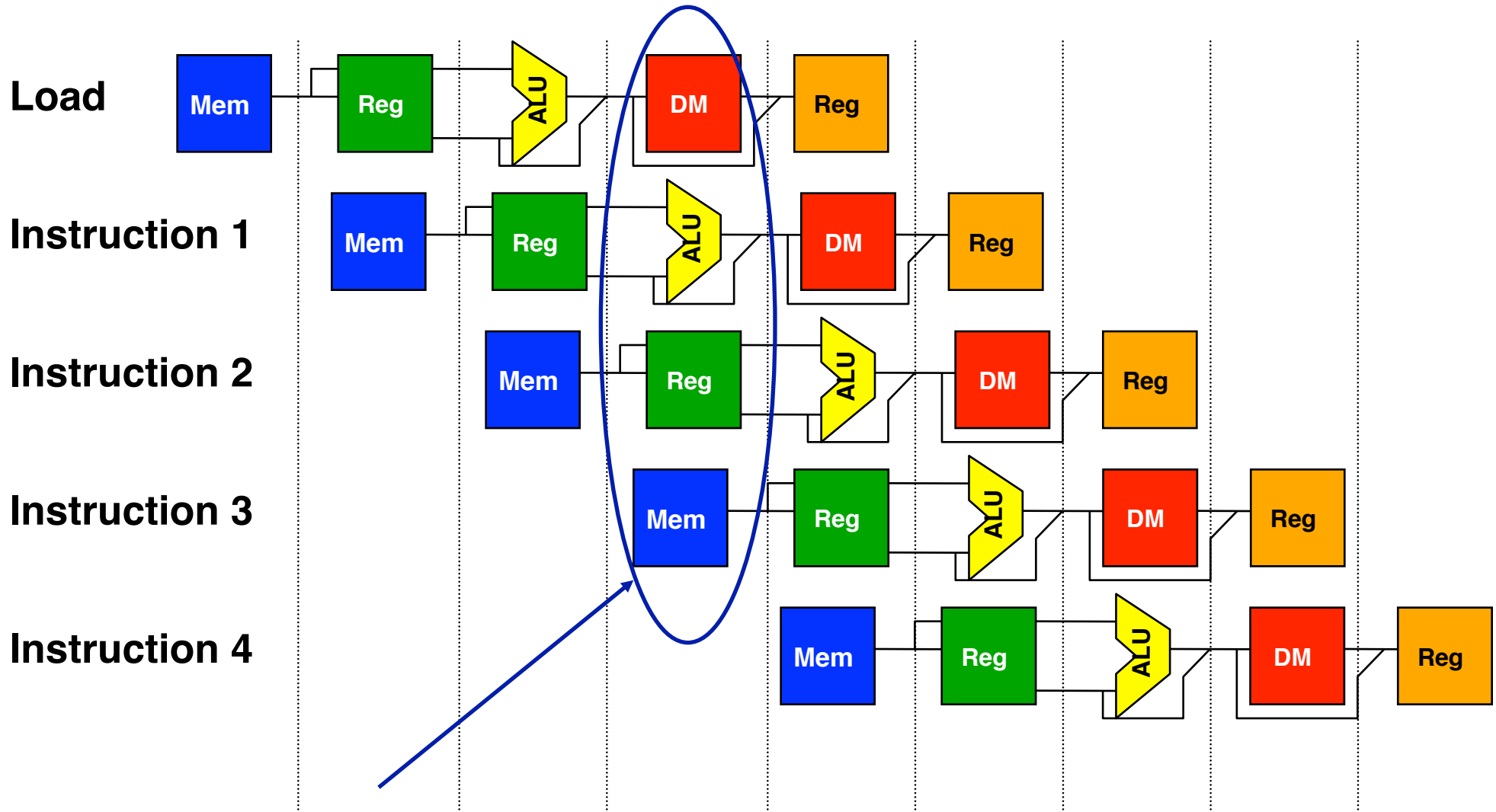
$$\textit{Speedup} = \frac{\textit{CPI unpipelined}}{1 + \textit{pipeline stall cycles per instruction}}$$

- If no stalls, speedup equal to # of pipeline stages in ideal case

# Structural hazards

- **Avoid structural hazards by duplicating resources**
  - **e.g. an ALU to perform an arithmetic operation and an adder to increment PC**
- **If not all possible combinations of instructions can be executed, structural hazards occur**
- **Pipelines stall result of hazards, CPI increased from the usual “1”**

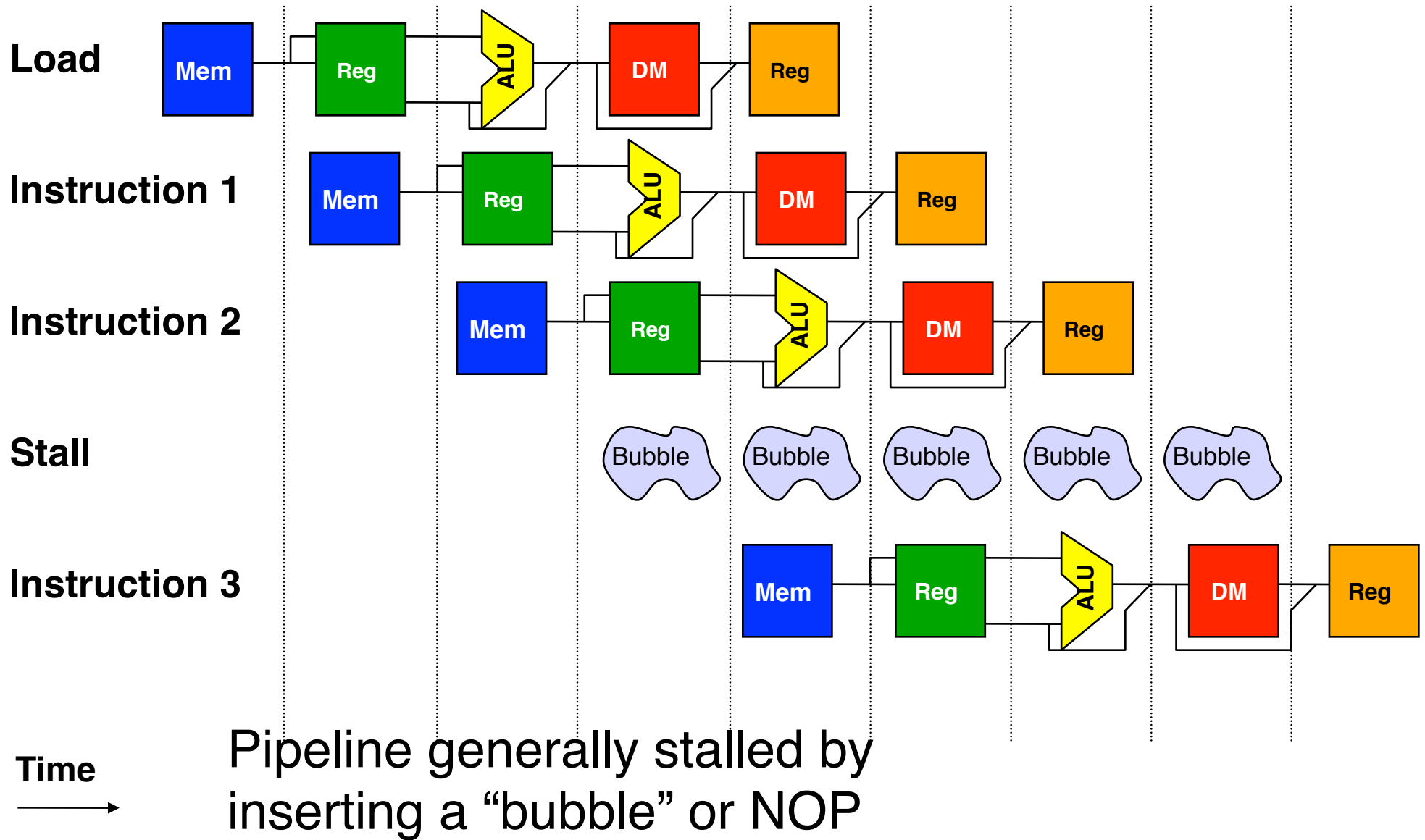
# An example of a structural hazard



What's the problem here?

Time  
→

# How is it resolved?



# Or alternatively...

← Clock Number →

Inst. #	1	2	3	4	5	6	7	8	9	10
LOAD	IF	ID	EX	MEM	WB					
Inst. i+1		IF	ID	EX	MEM	WB				
Inst. i+2			IF	ID	EX	MEM	WB			
Inst. i+3				stall	IF	ID	EX	MEM	WB	
Inst. i+4						IF	ID	EX	MEM	WB
Inst. i+5							IF	ID	EX	MEM
Inst. i+6								IF	ID	EX

- LOAD instruction “steals” an instruction fetch cycle which will cause the pipeline to stall.
- Thus, no instruction completes on clock cycle 8

# Remember the common case!

- **A machine without structural hazards will always have a lower CPI.**
- **But, in some cases it may be better to allow them than to eliminate them.**
- **Consider the following:**
  - **Is pipelining functional units or duplicating them costly in terms of HW?**
  - **Does structural hazard occur often?**
  - **What's the common case?**

# What's the realistic solution?

- **Answer: Add more hardware.**
  - **(especially for the memory access example!)**
    - **CPI degrades quickly from our ideal '1' for even the simplest of cases...**